Evaluating the relation between changeability decay and the characteristics of clones and methods

Conference or Workshop Item

Evaluating the relation between changeability decay and the characteristics of clones and methods

Angela Lozano, Michel Wermelinger, Bashar Nuseibeh
Computing Department and Centre for Research in Computing
The Open University, UK

Abstract

In this paper we propose a methodology to evaluate if there is a relation between two code characteristics. The methodology is based on relative risk*, an epidemiology formula used to analyze the effect of toxic agents in developing diseases. We present a metaphor in which the disease is changeability decay, measured at method level, and the toxic agent is a source code characteristic considered harmful. However, the formula assesses the strength of the relation between any toxic agent and any disease.*

We apply the methodology to explore cloning as a toxic agent that increases the risk of changeability decay. Cloning is a good agent to analyze given that although there is some evidence of maintainability issues caused by clones, we do not know which clones are harmful, or to what extent. We compare cloning with other possible 'toxic agents', like having high complexity or having high fan-in. We also use the technique to evaluate which clone characteristics (like clone size) may indicate harmful clones, by testing such characteristics as toxic agents. We found that cloning is one of the method characteristics that affects the least changeability decay, and that none of the clone characteristics analyzed are related with changeability decay.

1. Introduction

We present a metaphor to analyze the effect of source code characteristics, where the effect of a source code characteristic is equivalent to a disease, and the source code characteristic is the toxic agent that promotes the appearance of the disease. The goal is to analyze the strength of the relation between source code characteristics that are considered as bad implementation practices, and changeability decay (i.e. difficulty to change). However, the metaphor is also used to explore if there is any relation between source code characteristics (like being cloned and being a large or complex method), as well as to explore which clone characteristics are related with changeability decay. We propose cloning as a first characteristic to explore, given that empirical results have not been conclusive about its harmfulness.

Source code clones are the result of 'copy-paste' programming [1]. A clone is a source code fragment that is (nearly) identical to another fragment of code [2]. These similar fragments form a clone family (also called clone class [2]).

The effects of cloning are not yet fully known. Clones are believed to be harmful for several reasons [3]: clones need consistent changes, clones increase the source code size and creates hidden relations among fragments of source code, clones may indicate lack of abstraction, and finally clones may produce bugs when there are inconsistent changes or when there is inconsistent renaming of variables in the clone. However, some researchers have argued that there are circumstances in which clones do not have a harmful effect [1, 4]. In fact, we have shown, in a previous paper, that not all clones are harmful [5]. However, we do not know yet to what extent clones are harmful, or how to distinguish harmful from harmless clones.

Cloning is considered harmful, among other reasons, because of the need of changing all clones of a family in the same way, whenever any of them requires a change [3]. Such collective changes are called consistent changes [6], and inconsistent changes are believed to introduce bugs. There have been several attempts to measure the amount of consistent changes [6-9]. However, knowing that just 50% of the changes of cloned code are consistently replicated to the clone family [9] does not allow us to see to what extent clones are harmful. There can be other factors affecting the interpretation of this fact: for instance, if cloned methods change less than not cloned methods, then consistent changes may become less relevant. Although there is evidence that inconsistent changes introduce bugs [8], the study did not analyze all

inconsistent changes, so there is no indication of the rate of bugs due to inconsistent changes. Another example of the relation between clones and bugs, are the bugs generated whenever the pre/post conditions of the cloned fragments are not valid where the clone is pasted [10], but again that study fails to show to what extent cloning can be linked to bugs.

Nevertheless, none of these results help us, on one hand, to assess to what extent cloning is *more* harmful than other source code characteristics, e.g. overly high cyclomatic complexity, and on the other hand to identify characteristics that are related to the impact of the clone. Tackling these two issues will help us understanding the consequences of being cloned, and automatically point out among a large code base which clones are potentially harmful. Comparing cloning with other source code issues is also important because to be able to prioritize maintenance it is necessary to know which source code issues are the most harmful.

To measure the impact of cloning and other source code issues, we have chosen changeability decay, because changeability is a key factor of maintainability, the longest and most expensive phase in the software cycle, and therefore it increases the efforts for keeping the application in use. Changeability comprises the attributes of software that affect the effort needed for modifications, and changeability decay occurs when software characteristics hinder change. Changeability decay has been also defined as the increase of effort to implement and propagate a change [11]. Clones can affect changeability by increasing the propagation effort (consistent changes), and the number of changes (bug fixes).

In this paper, we propose a methodology to explore the existence of relations between source code characteristics, and the strength of such relations. We believe that correlation and regression may not show relations that are affected by many other factors. For instance, the attempt to correlate co-change and cloning at file level with different types of regression was unsuccessful [12]. Our method aims to be a lightweight approach to indicate relations, not causality.

The rest of the paper is organized as follows: section 2 describes the methodology proposed, section 3 explains the measurements we took and the relations we analyzed in order to understand better the effects of cloning, section 4 briefly describes the data gathering process, section 5 describes the results and the threats to validity, section 6 points to related work, and section 7 presents some concluding remarks.

2. Methodology

We propose to use a metric used in epidemiological cohort studies to assess whether a toxic agent would increase the risk that a person develops a disease [13], for instance to check if smoking increases the chances of developing lung cancer or not. The metric indicates how many times it is more likely to develop the disease if a person is exposed to the toxic agent. The metric is called relative risk (`RR`), and is defined as:

$$RR_{ToxicAgent\ Disease} = \frac{a/(a+b)}{d/(c+d)}$$

where `a` is the number of people that are sick and were exposed to the toxic agent, `b` is the number of people that are not sick but were exposed to the toxic agent, `c` is the number of people that are not sick and were not exposed to the toxic agent, and `d` is the number of people that are sick and were not exposed to the toxic agent. Having a relative risk higher than 1 means that the toxic agent increases the risk of developing the disease, while a relative risk lower than 1 means that the toxic agent decreases the risk of developing the disease. To avoid false positives, it is recommended to consider only relative risks higher than 2, and lower than 0.5 [14]. A relative risk higher than 3 indicates a strong association.

We chose relative risk because it is useful in cases where there is no control on the exposure to other toxic agents. That means that the formula indicates that there is a relation between the toxic agent and the development of the disease but also that there could be other toxic agents of greater risk to develop the disease. Given that cloning has been claimed to be harmful for maintenance, cloning could be considered as the toxic agent that increases the risk of developing changeability decay, which could be considered as a disease because it restricts the chances of the software system to 'survive'.

We propose to use relative risk to explore the strength of the relation between two characteristics: the agent (`A`) and the disease (`D`). Although relative risk is used for boolean characteristics (either the disease is manifest or not, and there has been exposure to the agent or not), we have adapted it to characteristics with numerical metrics by transforming the numerical metric into a boolean condition: to have or not a high (or low) value for that metric. We first define an agent to be high if its value is over the 75 percentile of the values of that agent in the application, and the agent is said to be low if its value is below the 25 percentile. Then we define a disease to be manifest if its value is

over the 75 percentile of the values of that characteristic in the application. Next, we compute the relative risk for four cases: RR_{HA_D}, $RR_{HA\_!D}$, RR_{LA_D}, $RR_{LA\_!D}$, where HA means high value in characteristic A, LA means low value in characteristic A, D means the disease occurs, and !D means it does not. One can then check that characteristic A is related to characteristic D if:

Case 0: one characteristic grows when the other one shrinks, and vice versa i.e., RR_{HA_D} and $RR_{LA\_!D}$ are lower than one, and $RR_{HA\_!D}$ and RR_{LA_D} are greater than one.

Case 1: both characteristics grow or shrink i.e., RR_{HA_D} and $RR_{LA\_!D}$ are greater than one, and $RR_{HA\_!D}$ and RR_{LA_D} are lower than one.

For instance, suppose that characteristic A is activity of white globules, and characteristic D is temperature. We would like to know if having fever (high temperature) is a good indicator of having an infection (high globule activity). Consider the following data set for 230 cases, of which 54 have fever, 59 have the top 25% white globule activity, and 63 have the bottom 25%:

	LA		HA
D	4	17	33
!D	59	91	26

! LA

!HA

With the above data, we have:

RR_{HA_D} = (HA_D/ HA) / (!HA_D / !HA)
= (33 / 59) / (21 / 171) = 4.5

$RR_{LA\_!D}$ = (LA_!D / LA) / (!LA_!D / !LA)
= (59 / 63) / (117 / 167) = 1.3

RR_{LA_D} = (LA_D / HA) / (!LA_D / !HA)
= (26 / 59) / (150 / 171) = 0.5

$RR_{HA\_!D}$ = (HA_!D/ LA) / (!HA_!D / !LA)
= (4 / 63) / (50 / 167) = 0.2

Given that RR_{HA_D} is greater than one, it means that when the temperature is high the risk of having high white globule activity is greater than when the temperature is not high; in fact, the risk of having high white globule activity is four times greater. Similarly, due to $RR_{LA\_!D}$ being greater than one, whenever the temperature is low the likelihood of having low white globule activity is more than twice, than when the temperature is not low. $RR_{HA\_!D}$ and RR_{LA_D} are lower than one, indicating that when the temperature is high it is unlikely that the activity of the white globules is low, and that when the temperature is low it is unlikely to observe high activity of white globules. Therefore, having this example corresponds to the **case 1** explained above, and we can say that the temperature is a good indicator of the white globule activity, because the temperature value follows the white globule activity.

Notice that it is necessary to check all the four possibilities of relative risk in order to claim that one characteristic is related to the other one. If the data set does not allow concluding either case 1 (direct relation) or case 0 (inverse relation), it means that characteristic A (agent) cannot predict characteristic D (disease).

Notice also that the value of relative risk gives an estimate of how much the risk of having the characteristic D increases. In order to keep this characteristic of relative risk, we propose to average the relative risks that are greater than one, only if case 0 or case 1 occurs. For example the relative value of the example shown previously would be 2.95 ([4.5+1.3]/2)

For our study on cloning and maintainability, the agent will be some characteristic of the methods or the clones within the methods, and the disease will be some characteristic that captures changeability decay. We will also relate the boolean characteristic of being cloned or not (A) with characteristics of the method (D). In the following section we explain in detail the chosen characteristics, how they were measured, and the rationale to explore such relations.

3. Experiment

In previous work [5] we found that methods cloned have different changeability decay measurements than methods not cloned, and that there are cloned methods that present a severe decay. However it is not clear whether removing clones should be a priority in maintenance, why having clones at method level may decrease the method's changeability, nor how to distinguish harmful clones from harmless ones. Therefore, this experiment aims to address three research questions.

(1) To what extent is cloning harmful compared to other source code characteristics also perceived as harmful?
(2) Are clones related with large or complex methods?
(3) What clone characteristics are related with changeability decay?

These relations are analyzed with the relative risk formula in order to assess if a characteristic is strongly

related to another characteristic. To address questions (1) and (2) we need method characteristics, explained in subsection 3.1. To address question (3) we need clone characteristics, explained in subsection 3.2. Questions (1) and (3) also require changeability decay to be characterized, as explained in subsection 3.3.

3.1. Method characteristics

For question (1) we compare the effect of a method being cloned, complex, large, and instable on the changeability decay measurements. For question (2) we check if being cloned is related to the size or complexity of a method.

The methods are characterized using metrics. A snapshot in the history of the application permits to extract the metrics of the methods that are part of the code base in that commit. However, a single snapshot is not enough to get metrics for all methods. Therefore, several snapshots are needed in order to cover all methods at least once. For our case studies, 3 to 4 snapshots were enough to cover all methods. Whenever a method had several values for a single characteristic these were averaged.

3.1.1. Being cloned. This is a boolean characteristic to say if the method has had a cloned fragment at any point in its lifetime or not.

3.1.2. Size. This characteristic is measured in lines of code (LOC).

3.1.3. Complexity. This characteristic is measured in two ways: using cyclomatic complexity which is the number of branches in the method; and counting the largest depth of block in the method (block depth).

3.1.4. Instability. The measurement of instability is defined as the fan-out divided by the sum of the fan-in and the fan-out of the method, i.e. an extrapolation of the instability measurement proposed by Robert Martin [15]. The fan-out could indicate if the method is too sensitive to changes in other methods or not, given that if depends on too many methods any change on them could affect it. The fan-in could indicate to what extent changing the protocol of a method affects the rest of the application.

3.2. Characteristics of clones

We want to know if the duration and size of clones, the number of hidden relations due to clones, or the distance between clones of the same clone family can be related to the effect of clones on changeability decay. However, since decay will be measured for methods (see next subsection), the clone characteristics will be also measured on the methods that contain clones, and, except for the duration metric, averaged over all commit transactions during which the method had clones.

3.2.1. Percentage of lifetime cloned. We analyze this characteristic to validate the intuitive argument that the longer a method is cloned, the worse is its changeability decay, because it has to maintain the hidden clone relations for longer. The characteristic is calculated as the percentage of commits in which the method had a clone over the number of commits in which the method was part of the code-base of the application.

3.2.2. Clone size. This characteristic helps to check to what extent it is true that cloning decreases changeability due to the loss of understandability that code bloating generates. This characteristic is defined as the average number of tokens of the largest clone inside a method, while the method was cloned.

3.2.3. Number of methods cloned with. We take into account this characteristic because given that clones require consistent changes, a higher number of cloning relations may decrease changeability. This characteristic is defined as the average number of methods with which the method shares a clone during the time while the method has a clone.

3.2.4. Distance to clones. This characteristic helps to assess to what extent the difficulty of finding the hidden relations due to cloning increases the number of changes required until all clones in a family are consistent again, and thereby increases changeability decay. We take the average number of directories for all clone families to which the method belongs. For instance, if the method belongs to two clone families of 2 clones each, and the other member of the first family is in the same file (distance = 0), and the other member of the second family is in the same directory but in a different file (distance = 1), then the method's distance to the other methods it is cloned with is 0.5.

3.3. Changeability decay measurements

The changeability decay measurements are used to address questions 1 and 3, where changeability decay is a disease that might be related to several source code characteristics as well as to several clone characteristics. Knowing which source code characteristics have more impact in changeability could help to prioritize maintenance tasks, while identifying which clone characteristics are linked to changeability decay could help to locate harmful clones.

Changeability decay is the increase of effort required to implement changes to the system. Ideally changes should be limited in quantity and in scope. The measurements presented aim to cover from different viewpoints these two aspects. To measure the quantity, we use amount of changes required (`number`) and periodicity of the changes (`frequency`). To assess the scope of changes, we define ripple effect (`impact`), modularity of the changes (`span`), and extension of the changes (`depth`). A higher measure indicates that either the method changed more or the propagation of its changes is more complex, hence the higher the measure the higher the decay.

It may seem pointless to have similar measurements for the same aspect, like frequency and number of changes. However, we have shown in previous analyses [16] that, although the number of changes seem to increase with clones, the likelihood (number of changes to a method divided by the number of changes) is not strongly affected by clones [5]. This happens because likelihood is a ratio measurement, and when a method is cloned both numerator and denominator increase. Therefore, although likelihood indicates that cloning has little effect on changeability, when looking at the number of changes one can confirm that such indication can be misleading.

For each method, the measurements are calculated for up to three periods: the lifetime of the method, when the method has clones, and when the method does not have clones. If a method never had clones, then the not cloned period coincides with its lifetime period, while its cloned period is empty. If a method always had a clone, its cloned period coincides with its lifetime period, while its not cloned period is empty. The measurements for the whole lifetime are used when addressing question (1), which requires comparison with method characteristics that are independent of cloning. The measurements for the period when cloned are used to address question (2), which is explicitly about cloning. Finally, the difference of the measurements between the cloned and not cloned periods are used to address question (3), to see whether changeability decay is different in the presence of cloning.

Table 1 summarizes the set of changeability measurements for a method *m* in a period, where a period is a set of commit transactions.

Table 1. Changeability decay measurements

Number=	No. of commits in which *m* changed during the period
Frequency=	$\frac{number}{\text{No. of commits in the period}}$
Impact=	$Avg. \frac{\text{No. of methods modified with } m}{\text{No. of methods alive}}$
Span=	Avg.. of packages modified
Depth=	*Avg. (**weight * distance***) ***weight*:** No. changed methods in *m*'s class / No. of methods changed ***distance:*** No. of packages away from *m*'s class to the closest package ancestor of the set of classes changed

In order to illustrate the changeability decay measurements, calculated on some method `getValue` during some period P, imagine the following situation. The application is composed of 100 methods throughout P. The period P is composed of 10 commits, which changed 2, 4, 3, 5, 2, 3, 1, 6, 2, 3 methods respectively. The method `getValue` was modified in the third and seventh commits. The first, third, seventh and ninth commit modified methods in classes `a.b.c.d.Class1` and `a.b.c.e.Class2`. Suppose that in the third commit changed one method (`getValue`) of the class `a.b.c.d.Class1`, and two methods of the class `a.b.c.e.Class2`.

3.3.1. Number. This measurement represents the number of changes the method requires. The number of changes of `getValue` is 2.

3.3.2. Frequency. This measurement shows how often the method requires changes. The frequency of changing `getValue` is 0.2, because it changed in two of the 10 commits that form the period.

3.3.3. Impact. This measurement represents the ripple effect or the average quantity of methods that require changes whenever the method changes. In the example, `getValue` was modified twice during the period: the third commit affected 3/100 = 3% of the application's methods, and the seventh commit affected 1% of the methods. The impact of `getValue` is hence (0.03 + 0.01) / 2 = 2%. This metric is a modification of the one presented in [17], and we provide more details in [5].

3.3.4. Span. This measurement expresses to what extent the changes are confined within a module or not. The span of the changes that affected `getValue` is 2 given that both the third and seventh commits affected two packages (`a.b.c.d` and `a.b.c.e`).

3.3.5. Depth. The depth measures the dispersion of the changes across different packages. If the distance between the methods changed is high, finding all the methods that the change requires may take longer. However if most of the methods changed are in a single class (weight) the distance should not affect significantly the effort of performing the change.

The distance can be computed as the number of tokens that are not included in the common prefix of the set of classes modified. The distance of the class `a.b.c.d.Class1` to the rest of the classes modified is 2 because the classes only share the prefix "`a.b.c`" but "`d`" and "`Class1`" are not in the shared package prefix. The weight of the class `a.b.c.d.Class1`, i.e. its share in the overall changes, is 1/3 because just one of the three methods changed in the commit belonged to that class. Therefore, the depth of the third commit is 6/3 = 2*1/3 + 2*2/3. The depth for `getValue` would be the average of the depths of the third and seventh commits.

4. Experiment set-up

We have already detailed our data collection approach in previous papers [5, 18]. This section only provides a brief summary.

The measurements presented in the previous sections are computed over a database that stores for each commit transaction the methods that were part of the application after each commit, and if they were modified or cloned by that commit transaction.

To populate the database, we wrote a tool to mine CVS repositories. In CVS there is no concept of atomic transactions and hence they have to be reconstructed from other information in the repository: to identify the commit transactions we use temporal proximity, same author and same commit message.

Given that the methods are moved across classes and packages, renamed, and change their parameters, we clean the data by performing origin analysis, i.e. by checking if a method identified as new is in fact a renamed/moved version of a previously existing method.

The cloning information is obtained by applying an automatic clone detection tool, CCFinder, after each commit transaction. CCFinder finds clones by string matching, and clones have to be at least 30 consecutive tokens long. CCFinder reports the pairs of token sequences cloned. Our tool translates this information into pairs of methods that shared a clone.

Table 2 presents the open source projects analyzed. GanttProject is a scheduling application with facilities for resource management. JEdit is a text editor that can be configured as an IDE through its plug-in architecture. FreeCol is game in which players have to conquer and colonize new worlds.

Table 2. Case studies

Project	KLOC	commits	Start month-end month	methods	methods cloned	methods cloned sometimes
ganttProj.	44	2701	May 03-Dec 06	11805	136	80
jEdit	92	1381	Sep 01-Jul 06	7392	346	159
freecol	54	1087	Apr 04-Mar 07	3901	310	159

5. Results

Regarding the questions analyzed with the proposed methodology, we found that:

1. There are several source code characteristics that increase the quantity of changes, like being: cloned, large, complex or instable. However, these characteristics also reduce the scope of changes, which means that these characteristics produce more isolated modifications. Besides, we could also find that there are source code characteristics that have a higher impact on changeability decay than cloning, e.g. the length, fan-out and complexity of the method.
2. No relation was found between being cloned and the size or the complexity of the method.
3. From the cloning characteristics analyzed, the only one related with changeability decay is the percentage of the method's lifetime that was cloned.

The rest of the section presents these results in detail.

Tables 3 and 5 present the relations between the method and clone characteristics (rows) and the changeability decay characteristics (columns). If the row measurement is directly related to the column measurement (i.e. **case 1**), the corresponding cell has the number 1. If the row measurement is inversely related to the column measurement (i.e. **case 0**) the cell shows the number 0. If there is no conclusive relation between the row and column measurement, the corresponding cell has a minus character. Each cell has one number or minus sign per case study, always in the same order: first freecol (f), then jedit (j), and finally ganttProject (g). Whenever the characteristics behave in the same way for all case studies, the cell is

shadowed: it is light when there is a direct relation and dark when there is an inverse relation.

Tables 4 and 6 present the average relative risk among the 3 case studies for the characteristics that were found to be good predictors. The values help to assess which characteristics predict better which maintainability behaviors. Those relative risk values that should be considered (i.e. are greater than 2) are emphasized in bold.

5.1. Relation between method's characteristics and changeability decay

As Table 3 shows in the light gray columns, quantity measurements (number, frequency) follow most of the characteristics (except instability and fan-in). However, another scope measurement (impact) shrinks when the characteristics grow, as the column in dark gray shows. This would mean that cloned, large, complex, or instable methods tend to increase the need of changing the method. But, such changes are performed in isolation.

Fan in is directly related with impact (scope) and number (quantity). That would mean that having a high fan-in would increase the changeability decay, which is counter intuitive given that a high fan-in shows reuse. Nevertheless, table 4 shows that the relation of fan-in with impact and number of changes is rather accidental, given that its relative risk is very close to 1. Therefore, although a high fan-in does not increase the changeability measurements, it does not decrease them either.

Table 3. Relation between method's characteristics and changeability measurements (question 1)

	Nm.	Frq.	Imp.	Sp.	Dep.
	f j g	f j g	f j g	f j g	f j g
Being cloned	111	111	000	011	011
LOC	111	111	000	0-1	111
Cyc. complex.	111	111	000	0-0	-11
Block depth	111	111	000	011	001
Instability	11-	111	000	110	110
Fan-in	111	00-	111	00-	00-
Fan-out	111	111	000	011	111

According to the values in table 4, the characteristics that affect changeability, from the most related to the least related, are the lines of code, the fan-out, the block depth, the cyclomatic complexity and being cloned.

The number of lines of code (LOC) increases with the quantity of changes. Notice that LOC is the only characteristic inversely related to the impact (2.0): that means that the longer a method is, the more likely it is to be changed in isolation.

A high fan-out, block depth, or cyclomatic complexity increases the chance of having high quantity of changes. However, contrary to expected, the relation with quantity measurements is stronger for the lines of code (LOC) and the fan-out, than for cyclomatic complexity.

Being cloned is directly related to the quantity of changes. Being cloned is in third place on the characteristics directly related to the number of changes (2.36). That means that although being cloned increases the chance of having a higher number of changes, that chance is higher when the method has high fan-out or when it is large.

The instability does not seem to have any relation with the changeability measurements.

Table 4. Relative Risk average for each method characteristic on changeability measurements

	Nm.	Frq.	Imp.	Sp.	Dep.
Being cloned	**2.3**	1.7	1.3		
LOC	**2.5**	**2.9**	**2.0**		1.2
C. complex.	**2.1**	**2.2**	1.6		
Block depth	**2.0**	**2.0**	1.8		
Instability		1.6	1.6		
Fan-in	1.1		1.1		
Fan-out	**2.5**	**2.8**	1.7		1.2

Summarizing, methods that are large, complex or in charge of many responsibilities are more likely to cause more changes to the application. Although being cloned increases the number of changes, the length, fan-out and complexity of the method influence more the quantity of changes.

5.2. Relation between cloning and method's size and complexity

For none of the case studies the relative risks of being cloned versus the size and complexity of the method complied with cases 0 or 1. Therefore, being cloned is *not related* with the size or the complexity of a method. Given that there is no point in showing a table full of '-' characters, there is no table to present these results.

5.3. Relation between cloning characteristics and changeability decay

The analysis of cloning characteristics intends to explore alternative ways to identify harmful clones. The results are shown in tables 5 and 6. The upper part of these tables show relative risks between the clone characteristics of the methods sometimes cloned, and the increase on changeability decay measurements when cloned (i.e. the difference of the measurements when cloned and when not cloned). The lower part of these tables show the relative risks between the clone characteristics of the methods that were cloned at least during one commit, and the changeability decay measurements in the period when they were cloned.

Table 5. Relation between cloning characteristics

	Nm.	Frq.	Imp.	Sp.	Dep.
	f j g	f j g	f j g	f j g	f j g
Lifetime cloned	111	000	001	0-1	111
Cloned with	-11	00-	10-	00-	-11
Clone size	--1	110	-1-	1-1	---
Distance	101	0-0	0-1	0-1	111
Lifetime cloned	110	000	001	10-	1--
Cloned with	011	000	100	000	00-
Clone size	011	-0-	-01	---	11-
Distance	-0-	--0	000	111	1-1

Table 6. Relative Risk average for cloning characteristics in changeability measurements

	Nm.	Frq.	Imp.	Sp.	Dep.
Lifetime cloned	2.2	3.9			1.5
Distance					1.2
Lifetime cloned		2.9			
Cloned with		1.4		1.2	
Distance			1.4	1.6	

The upper part of each table shows that there is a direct relation between the lifetime and the number of changes which is rather obvious: the longer a method is cloned, the higher is the chance of having an increased number of changes when cloned.

The results regarding frequency are more interesting. It seems that the longer a method is cloned, the smaller is its frequency of change compared with the period not cloned. That is coherent with previous results [16] that indicate that the number of changes can be misleading.

The lower part of each table confirms that the longer a method is cloned the lower it is its frequency of change.

Summarizing, from the cloning characteristics analyzed, the only one that is related with changeability measurements is the percentage of the method's lifetime that was cloned. However, this result cannot be used to distinguish potentially harmful from harmless clones a priori, i.e. when they are introduced in the application.

5.4. Identification of clones harmful to changeability

Given the previous result, we decided to mix method characteristics and cloning characteristics in order to explore ways to detect harmful cloning situations. We chose the method characteristics that had the strongest direct or inverse relation to changeability measurements (complexity, LOC, fan-out), and combined them with the cloning characteristics that can be calculated a priori (cloned with, clone size, distance). Only the combination of distance and fan-out led to the same type of relation (namely case 1) in the three case studies. However, the relative risk was 1.44, and could hence be a false positive. Again, there is no table to present these relative risks given the lack of results.

6. Threats to validity

One threat of the methodology is the choice of thresholds. Even if the sample has only large values or only low values, there will be always low and high percentiles. Therefore, threshold values could differ across case studies. However, this problem is addressed by taking into account only those results in which all case studies showed the same behavior. If the behavior of the case studies differs it is possible to choose the same thresholds for all case studies to reflect large and small values for a characteristic, this analysis will be covered in future work.

Another issue is that the structural metrics are calculated based on the static source code. That means that metrics like fan-in may be miscalculated due to missing late binding information. Such lack of accuracy may have influenced the results.

The approach to assess changeability decay also poses problems for the interpretation of the results.

Given that the changeability decay is described by several measurements, it is not clear what the overall effect is.

Another issue is that the interpretation of the measurements is linked to the assumption that the changes analyzed represent sets of methods that are related in accomplishing a feature. In order to discard such assumption, we eliminated those changes that seem to be restructurings and affected most of the methods of the application. However, other types of atypical changes, for instance committing by time intervals instead of committing when finishing a change, may also affect this assumption, and therefore the interpretation of the measurements.

Finally, although taking into account those situations in which all case studies had the same type of relation (i.e. all case 1 or all case 0) makes the result stronger, the methodology becomes weaker as the results depend on the amount of case studies used.

7. Related work

Besides the work mentioned in the introduction about cloning, there also have been several attempts to measure changeability decay. This paper aims to find structural predictors of as many dimensions as possible of changeability decay, in contrast to previous work that proposed and evaluated changeability decay measurements. Bianchi et al. [19] present an empirical experiment in which the degradation of the architecture in assessed in terms of entropy, which is the increase of disorder in the traceability links of a software system. They found that that the connectivity and complexity inside and across components increased with changes. Eick et al. [20] defined several ways to know if a change is more difficult than what it should be and called it decay. They found that the modularity decreases, the number of files touched increases, new changes introduce bugs, and time to perform changes increases, indicating changeability decay. Arisholm et al. [11] compared change and structural measurements using regression models to assess which one determined changeability decay better. They found that change measurements capture changeability decay dimensions that structural measurements cannot capture.

There are also some studies aiming to analyze the impact of source code characteristics. However, none of them tried to compare the impact of diverse source code characteristics. Ratiu et al. found that god classes do not always have bad impact given that some of them are stable (do not change) and others of them are volatile (have a reduced lifetime) [21].

Finally, Ueda et al. define clone metrics similar to ours [22]. For instance, the length of the clone class is similar to the clone size, and the clone radius is similar to the distance to clones. The length of the clone class is the number of consecutive tokens shared by all the clone fragments that belong to the same clone family, while the clone radius is the maximum number of directories to the nearest common ancestor of the files that share the same clone. However, neither the clone length nor the clone radius are applicable at method level.

8. Conclusions

We proposed measurements to assess changeability decay, in addition to those proposed previously. We have proposed a novel perspective on the analysis of source code characteristics by using a biological metaphor that likens source code characteristics issues to toxic agents, and changeability decay to a disease. The methodology assesses the strength of the relation between two source code characteristics. Using this methodology we have shown that for all case studies:

- Cloning is not strongly related with the complexity or size of a method.
- The percentage of lifetime cloned does not increase the changeability decay measurements; in fact it seems to decrease some of them (frequency).
- Although being cloned increases the risk of having changeability decay, there are other characteristics of a method that have a greater impact on changeability decay. This could mean that managing cloning should not be the first priority in maintainability. This could also mean that cloning is just another symptom of some implementation ‘disease’.
- Of those clone characteristics analyzed, none of them can be used to a priori identify those clones harmful to changeability.

The method proved to be lightweight and useful form to evaluate if a characteristic is a good predictor of another characteristic.

9. References

[1] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in OOPL," in *Proc. of the Int'l Symp. on Empirical Software Engineering*, 2004, pp. 83-92.

[2] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 654-670, 2002.

[3] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," in

Proc. Int'l Conf. on Software Maintenance, 1999, pp. 109-118.

[4] C. Kapser and M. Godfrey, "'Cloning considered harmful' considered harmful," in *Proc. of the Working Conf. on Reverse Engineering*, 2006, pp. 19-28.

[5] A. Lozano and M. Wermelinger, *"Assessing the effect of clones on changeability" to appear in Proc. Int'l Conf. on Software Maintenance*, 2008.

[6] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *Proc. of the European Softw. Eng. Conf. and symp. on Foundations of Softw. Eng. (ESEC-FSE)*: ACM Press, 2005, pp. 187-196.

[7] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl, "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process," in *Proc. of the Int'l Conf. on Software Maintenance*, 1997, pp. 314-321.

[8] L. Aversano, L. Cerulo, and M. D. Penta, "How Clones are Maintained: An Empirical Study," in *Proc. of the European Conf. on Software Maintenance and Reengineering*, 2007, pp. 81-90.

[9] J. Krinke, "A Study of Consistent and Inconsistent Changes to Code Clones," in *Proc. of the Working Conf. on Reverse Engineering*, 2007, pp. 170-178.

[10] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*. Banff, Alberta, Canada: ACM, 2001, pp. 73-88.

[11] E. Arisholm and D. I. K. Sjoberg, "Towards a framework for empirical assessment of changeability decay," *J. Syst. Softw.*, vol. 53, pp. 3-14, 2000.

[12] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger, "Relation of Code Clones and Change Couplings," in *Proc. of the Int'l Conf. of Fundamental Approaches to Software Engineering*, 2006, pp. 411-425.

[13] M. Green, M. Freedman, and L. Gordis, "Reference Guide on Epidemiology," in *Reference Manual on Scientific Evidence*, 2 ed: Federal Judicial Center: Washington, DC, 2000, pp. 638.

[14] G. Taubes and C. Mann, "Epidemiology faces its limits," in *Science*, 1995, pp. 164-169.

[15] R. C. Martin, "Stability," in *The C++ Report*, 1996.

[16] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the harmfulness of cloning: a change based experiment," in *Proc. of the int'l workshop on Mining Software Repositories*: IEEE Computer Society, 2007, pp. 18-21.

[17] T. B. V. Belle, "Modularity and the Evolution of Software Evolvability," The University of New Mexico, 2004.

[18] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Assessing the impact of bad smells using historical information," in *In Proc. of the Int'l Workshop On Principles of Software Evolution*: IEEE Computer Society, 2007, pp. 31 - 34.

[19] A. Bianchi, D. Caivano, F. Lanubile, and G. Visaggio, "Evaluating software degradation through entropy," in *Proc. of the int'l symp. on Software Metrics (METRICS)*: IEEE Computer Society, 2001, pp. 210-219.

[20] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data," *IEEE Trans. Softw. Eng.*, vol. 27, pp. 1-12, 2001.

[21] D. Ratiu, S. Ducasse, T. Girba, and R. Marinescu, "Using History Information to Improve Design Flaws Detection," in *Proc. of the European Conf. on Software Maintenance and Reengineering*, 2004, pp. 223-232.

[22] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, "Gemini: maintenance support environment based on code clone analysis," presented at Proc. Int'l Symp. on Software Metrics, 2002.